On Energy, Adaptation, and the Death of Frames

Albert F. Harris III Robin Snader Robin Kravets
Department of Computer Science
University of Illinois at Urbana-Champaign
email: {aharris, rsnader2, rhk}@cs.uiuc.edu

ABSTRACT

Increases in computing capabilities of mobile devices have led to the use of multimedia applications that have high processor and bandwidth resource requirements, each of which consume significant energy. However, battery capacities have not kept pace, driving the design of energy-aware applications. Traditionally multimedia applications have used encoding techniques to deal with bandwidth constraints, essentially trading off CPU for networking resources. However, in the context of energy constraints, it is not clear that this CPU-intensive approach is the most efficient. Making correct trade off decisions requires information about costs of both the CPU and the network. In addition to energy-savings by adaptive applications, further energy conservation can be achieved by leveraging application layer information at the network layer. Because of their ability to tolerate loss, multimedia applications present unique opportunities to the design of such energy-efficient network protocols, despite their strict timing constraints. Essentially, transport protocols can use application level information to make intelligent decisions about when to perform frame recovery, positively impacting system energy conservation. Previous adaptive systems concentrate on sharing resource information between the application and the network in only one direction and so perform sub-optimally. Therefore, it is necessary to design cooperative solutions that share resource information in both directions. To this end, we present a data-oriented energy model that exposes cross-layer interactions and enables specific optimizations in the application and network layers. We demonstrate that by passing a minimal amount of information in both directions, an adaptive application paired with our adaptive transport protocol, Reaper, can achieve significant energy savings, which we verify through a system implementation.

1. INTRODUCTION

Current mobile devices are rapidly becoming more powerful, in terms of both processing power and network bandwidth. As the capabilities of these systems grow, applications conventionally associated with non-mobile devices are being moved into the mobile domain. However, these capabilities come at the cost of increased energy consumption, and battery capacity is not keeping pace with the growth in demand. Therefore, mobile systems are increasingly constrained by limited energy resources, driving the design of energy-aware applications and protocols.

Since energy consumption is closely tied to the amount of data transmitted, using cross-layer information in both applications and network protocols opens the door for techniques that can maximize energy efficiency. First, applications can leverage information about the costs of various system components (*e.g.*, CPU, network) to adapt to minimize energy consumption. Second, protocols can leverage information about the application's data to make decisions affecting energy consumption (*e.g.*, how much reliability is needed?). These protocol decisions, however, impact the cost of the network, which must then be filtered back up to the application. Therefore, a bi-directional information flow must exist between the application and the transport layer to support energy-aware applications and protocols.

Traditionally, multimedia applications have been primarily constrained by a lack of bandwidth. To deal with this constraint, techniques are used to encode raw video frames, reducing their size, before transmitting them across the network. These encoding algorithms leverage abundant CPU resources to compensate for limited bandwidth. However, in the context of energy constraints, it is not clear that this CPU-intensive approach is the most energy efficient. Additionally, current wireless networks can sustain bandwidths that do not always require the use of the most aggressive encoding techniques. Therefore, trading off arbitrarily high encoding costs to minimize transmission costs is not always the correct decision. However, making correct trade-off decisions requires information about about the per-cycle CPU costs and the per-byte network costs. Such information enables energy-efficient multimedia applications to adapt their encoding schemes according to the relative costs of the CPU and the network, doing less encoding and transmitting more bytes when the CPU is more expensive and doing more encoding and transmitting fewer bytes when the network is more expensive. Therefore, energy efficiency can be achieved at the application layer through the use of specific cost information about the network and the CPU.

While the costs of the CPU are completely determined by internal system parameters, the costs of the network are variable, changing with the prevailing channel conditions. Additionally, given information about the type of network service required by an application, the network may be able to reduce per-byte energy consumption. Essentially, the network costs depend on the amount of data transmitted, the characteristics of the channel and the needs of the application. Multimedia applications present unique opportunities for the design of such energy-efficient network protocols due to their ability to tolerate some loss and their strict timing requirements. Specifically, a transport protocol can use application-level information, such as video frame deadlines and reliability requirements, to make intelligent decisions about whether frame recovery should be performed, resulting in a reduced per-byte energy cost. In this paper, we define two components of the per-byte energy cost that are directly related to the characteristics of the application data. The first is the overhead imposed by the transport layer in terms of number of protocol bits transmitted per application bit. We call this the *data expansion*, which captures overhead such as the number of retransmissions, and so is dependent on the reliability requirements of the application. The second component is the actual goodput of the data (*i.e.*, the number of good frames at the receiver). Essentially, any frame that is transmitted by the sender that cannot be used by the receiver (*e.g.*, because it arrived late) constitutes wasted energy by the transport layer and adds to the overall per-byte energy cost. To minimize the number of late frames, the transport layer needs information about data timing requirements from the application.

Previous adaptive systems concentrate on sharing resource information between the application and the network in one direction or the other [10, 5], ignoring the interdependencies. In comparison, our research supports cooperative solutions that share resource information in both directions. To this end, we first evaluate energy consumption for wireless multimedia applications, exposing the explicit cross-layer interactions and enabling specific optimizations in the application and network layers. Using this model, we develop a transport layer, called Reaper, that leverages application information to minimize data expansion while providing effective network service to the application. Through extensive evaluations on a real system, we show that Reaper outperforms protocols that share information in one direction only. Finally, we present an adaptive video encoding application, called aVE, that uses the mechanisms provided by Reaper to dynamically adapt per-frame encoding to minimize the total system energy consumption. This application also demonstrates the ease of passing relevant information back to the transport layer for use in energy-efficient protocol design.

The rest of this paper is as follows: Section 2 presents our energy model and exposes the interrelationships between the network and the application. Section 3 discusses loss recovery mechanisms available to the transport protocol and discusses their impact on saving energy. It also presents three mechanisms for exploiting multimedia's deadline sensitivity and loss tolerance. Section 4 presents the design and implementation of Reaper, our energy-aware transport protocol. This description is followed by evaluations of Reaper's ability to maximize the number of frames successfully delivered on time while minimizing data expansion, and so energy consumption. Section 5 presents aVE, our adaptive video encoding application that demonstrates the use of bi-directional cross-layer information sharing to minimize the energy associated with generating and transmitting video frames. Finally, Section 6 presents conclusions and suggests directions for future research.

2. ENERGY CONSERVATION FOR MULTIMEDIA APPLICATIONS

Energy is consumed by all components of a mobile system, including the CPU and the network. However, there are interesting dependencies between these components that impact the energy consumption of the whole system. A good model of these dependencies can enable effective energy conservation algorithms that result in reduced total system energy consumption. Since the focus of our research is to support energy conservation through information sharing between adaptive applications and the network, we present an energy model that can then be used by the application and the network to drive their adaptations.

Since the goal of our target applications is to transmit multimedia data from a sender to a receiver, our energy model only considers energy consumption directly related to the handling of the application's data. Multimedia data (*e.g.*, video frames) is typically too large to transmit raw, therefore, multimedia applications encode the data to reduce its size. From the application's perspective, there are two main components of energy consumption: frame encoding and transmission. Therefore, per-frame energy consumption for the application can be defined as follows:

$$E_{frame} = E_{encode} + E_{trans}.$$

E_{encode} is determined by the number of CPU cycles needed to encode the frame (C), the speed of the processor in terms of processor frequency (F), and the amount of power to run the processor at that frequency (P_{CPU}):

$$E_{encode} = \frac{C}{F} \times P_{CPU}.$$

The encoding of the D bytes of the raw frame results in a new D'-byte encoded frame, where $\frac{D-D'}{D}$ is the encoding efficiency of the encoding algorithm. Many encoding algorithms support multiple schemes for encoding, each with their own encoding efficiency and CPU energy requirements, enabling a simple tradeoff between the amount of computation, C, and the resulting reduction in frame size, $D - D'$. An application can use information about these tradeoffs to adapt to changes in network conditions. However, without knowledge of E_{trans}, adaptations cannot optimize for energy consumption.

Once the application has encoded a frame, it is the job of the transport layer to send that frame. Since it must at least add headers to the data, the transport layer always imposes some overhead, and D'' bytes are actually transmitted per D' application bytes. $\frac{D''}{D'}$ represents the expansion factor for the transport protocol. Therefore, the energy to transmit the D' bytes is defined by this expansion factor, the transmission rate of the interface card (R), and the amount

of power to keep the interface card transmitting (P_{card}):

$$E_{trans} = \frac{D''}{R} \times P_{card}.$$

Clearly, the per-application-byte component of E_{trans} is determined by R, which affects how long a transmission takes, and P_{card}, which can be adapted based on the on the quality of the channel [7, 13]. While layering constraints dictate that applications should not know the low-level details about network headers or how the network adapts to changes in the channel quality, the application does need to know about the impact of these adaptations on the transmission rate and per-application-byte energy costs.

Although multimedia applications can often tolerate some loss, excessive losses may detrimentally affect the quality of the stream. Additionally, since the application has already consumed some amount of energy to encode the frame, not recovering from loss may waste that energy. To recover from loss, the transport protocol can add redundancy into the data stream, increasing D'', and so increasing E_{trans}. Again, although the application does not want to know the details of loss recovery, the impact of loss recovery on per-application-byte energy costs must be exposed.

In the next section, we examine this loss resilience, and discuss various mechanisms for achieving reliability and their impact on energy. We also discuss how selective unreliability can actually increase both the perceived quality of the multimedia stream, and the deadline-adjusted overall reliability of the stream as a whole, while increasing energy efficiency.

3. ON THE TOLERANCE OF LOSSES

One of the primary components of a network service is to provide sufficient loss recovery mechanisms to support the needs of the applications. Such loss recovery is achieved by increasing the number of bytes transmitted, either by adding redundant data to allow the receiver to reconstruct the original application data itself, as in the case of forward error correction (FEC), or more directly by retransmitting lost application data. However, loss recovery comes at the cost of increasing the energy consumed to transfer the application data. The additional energy investment necessary for reliability creates a tradeoff between perfect reliability and energy efficiency. However, these traditional loss recovery mechanisms that perform optimally in the absence of energy constraints perform poorly when confronted with such energy constraints [23]. We therefore evaluate these mechanisms to determine their impact on energy consumption.

Since application data units (*i.e.*, frames) are often much larger than the maximum packet size of the network, a transport protocol typically divides them into fragments. These fragments are then transferred across the network and reassembled before delivery to the application. Each packet that is lost in the network carries one of these fragments, and so fragments are the unit used for loss recovery. For a multimedia frame to be useful to the receiving application, all of its fragments must be received and reassembled before the frame's deadline. Since perfect loss recovery is expensive, and is usually unnecessary in the case of multimedia data, suspending loss recovery in certain cases can save energy and bandwidth. In this section, we first present two typical loss recovery mechanisms, FEC and retransmissions, and evaluate their effect on energy consumption and reliability in the face of timing constraints. Given that neither approach can provide energy-efficient 100% reliability and that multimedia applications may not require such reliability, we present two scenarios when loss recovery should be suspended and the subsequent impact on energy consumption.

3.1 Loss Recovery

As a proactive approach, FEC achieves loss recovery by adding some small amount of redundancy to each fragment of the transmitted data. In this case, $D'' = D' \times (1 + \mu)$, where μ is the amount of redundancy added per frame and so defines a correcting threshold. Since creating the error correcting codes requires computation and so incurs an energy cost, E_{comp}, the total energy cost for FEC-based loss recovery is

$$E_{FEC} = \frac{D'(1+\mu)}{R} \times P_{card} + E_{comp}.$$

FEC can only recover if the number of lost fragments remains below the correcting threshold. The larger the value of μ, the larger the correcting threshold and the more lost fragments can be recovered. The main benefit of such proactive approaches is immediate data recovery at the receiver. In the face of a loss, no time is wasted on retransmissions. The cost of FEC comes from the extra $\mu D'$ bytes added to each frame and from E_{comp}. If the number of lost fragments is below the correcting threshold, bandwidth and energy are wasted in the transmission of the extra redundancy. On the other hand, if μ is too small, loss recovery will fail and backup mechanisms will have to be used to repair the data stream, increasing E_{FEC}.

Retransmission-based loss recovery reactively recovers from loss by retransmitting lost fragments only when a loss is detected. Such approaches have the advantage of increasing the number of bytes transmitted, and so consuming extra energy, only when there is a loss. Therefore,

$$D'' = D' \left(1 + \frac{n}{N}\right),$$

where N is the number of fragments and n is the number of retransmissions of fragments needed to recover the data. Since there is little computational overhead for retransmitting fragments, the total energy cost for retransmission-based loss recovery is

$$E_{retrans} = \frac{D'(1+\frac{n}{N})}{R} \times P_{card}.$$

However, since losses typically cannot be detected before a round trip time has elapsed, it is possible that any retransmissions will arrive too late to repair the data, wasting bandwidth and energy on the unusable retransmissions. While it is obvious that retransmission-based recovery can cause the frame being repaired to be late, such techniques can also make subsequent frames late, wasting even more energy.

To determine which method is most energy efficient, it is necessary to compare the expected energy costs for the target environment. When the FEC code matches the loss rate of the channel exactly, μ is equivalent to $\frac{n}{N}$, since all of the

redundancy added by the FEC is used for repair. Even in this case, however, the overhead of E_{comp} makes FEC more expensive in terms of energy than a retransmission-based strategy. Additionally, in bursty environments like the Internet and wireless networks, the loss fraction $\frac{n}{N}$ varies, making it impossible to match the FEC error correcting ability to the loss rate. This results in either extra redundancy or overhead from using other repair techniques like retransmissions. By overestimating the loss rate, FEC can work well in wired environments (either in conjunction with application frame encoding techniques [22] or in multicast environment [12, 16]) where energy constraints are not an issue. In such environments, the rapid recovery outweighs the extra overhead for unused redundancy, which has little impact on performance. However, in wireless environments where energy is a major constraint and network conditions vary rapidly, the overhead for FEC is too high.

While FEC-based loss recovery is not effective in energy-constrained applications, neither is it the case that retransmission-based loss recovery is a perfect solution. When a retransmission arrives at the receiver in a timely fashion, the fragment is still beneficial to the application and so the energy overhead of the retransmission was not wasted. However, if the retransmission arrives too late or was not necessary, the energy put into the retransmission was wasted. This begs the questions of when, if ever, retransmissions should be attempted.

3.2 When a Frame Isn't Worth Saving

There are two unique characteristics of multimedia data that make retransmission-based loss recovery challenging. First, since multimedia applications have strict timing requirements, any frames that arrive past their deadlines waste energy. Early identification of these frames can save both energy and bandwidth. Second, the ability to tolerate some threshold of loss opens the possibility of opportunistically suspending loss recovery to save energy. In this section, we discuss both of these characteristics and their impact on energy consumption. In the following section, we show that while each of these approaches in isolation can save some minimal amount of energy, the benefits of combining them is greater than the sum of the individual improvements.

3.2.1 Timing Violations

Arriving frames that violate timing constraints cannot be used by the application and so all the energy that went into encoding and transmitting those frames is wasted. Although it is not possible to entirely eliminate such waste, two approaches can be used to reduce it. *Reactive* methods let the receiver notify the sender when a frame or a fragment of a frame is received late. The sender can then avoid sending any further fragments of that frame [17]. *Predictive* methods allow the sender to predict if a frame or a fragment of a frame will be late. The sender can then avoid sending anything that is predicted to be late. An analysis and design of this method is one of the contributions of our work.

Reactive methods leverage the fact that the receiver knows the deadline of a frame. Therefore, the receiver can tell if that deadline has passed during the reception of the fragments of a frame. The receiver can then signal the sender to stop sending any further fragments of that frame [17]. For example, the last two frames in Figure 1 could be dropped at the sender. While this can save the transmission energy of the unsent fragments, the energy to encode the frame and the energy used to transmit the initial fragments is wasted. Additionally, the ability of this mechanism to save energy by preventing the sender from transmitting useless fragments of a frame depends on the size of the frames, the bandwidth, and latency of the link. If all of the fragments of a frame are in flight before the drop message is received by the sender, this mechanism results in no energy savings (see Figure 2).

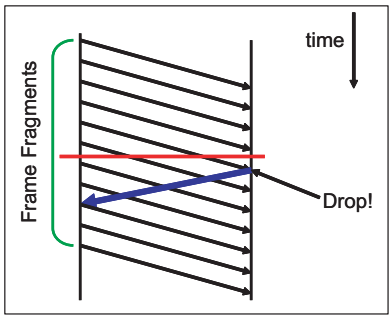

Figure 1: Receiver Late Frame Mechanism

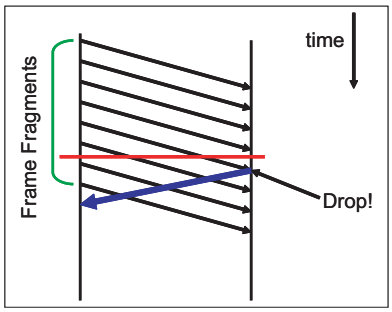

Figure 2: Receiver Late Frame Mechanism Failure

Predictive methods leverage the fact that the sender has some information about frame deadlines and estimations of the round trip time (RTT). Therefore, the sender can predict the one-way latency and so predict whether a frame will arrive late. Similar to the reactive methods described above, such predictive methods can save the energy from transmitting unsent fragments. Additionally, if the prediction is made early enough, it may be possible to signal the application not to encode the frame, hence saving all energy associated with the frame. The challenge to predictive methods is that they require accurate estimations of network latency. If the estimation algorithm is too pessimistic, frames that could have arrived on time may get dropped, and if it is too optimistic, frames that can never arrive on time may still get transmitted.

Both methods are limited by the relationship between the network latency and the frame deadlines, and so applications with timing constraints can be made more robust to timing violations by the use of a playout buffer at the receiver to increase the amount of time available for retransmissions [11]. We discuss the impact of such a playout buffer on loss recovery in Section 4.1.

3.2.2 Opportunistic Dropping

It is obvious that suspending loss recovery for frames that are already late or predicted to be late can save energy and bandwidth. However, it is also possible to leverage information about current performance (in terms of the fraction of frames successfully delivered) to decide whether to suspend loss recovery even if the frame might arrive on time, improving energy efficiency and successful frame delivery rate.

If the receiver is not experiencing many dropped frames, it is possible that the application could tolerate the loss of a frame. Given an application-specified reliability threshold, it may be possible to save energy by opportunistically dropping a frame with lost fragments and avoiding the retransmissions. This approach has two effects. First, energy that would have been expended in the retransmission of the lost fragments is saved. Second, the stream can catch up in time by skipping ahead to the next frame, potentially even preventing subsequent frames from being late. Additionally, since packet loss frequently indicates congestion in the network, it may be the case that a retransmission would be lost anyway, and that trying to force the frame through would simply be throwing good energy after bad.

If the receiver has been tracking the quality of the received data stream, the receiver may determine that the current quality is above the application's threshold. A smart receiver could then decided to not ask for retransmission of the lost fragments [8], saving energy from the retransmission [6]. This can be implemented by allowing the receiver to send a false acknowledgment of lost fragments when the quality is higher than necessary. We take this approach one step further and allow the receiver to send a false acknowledgment for the whole frame, allowing the sender to avoid sending any unsent fragments in the frame.

In the next section, we discuss how these techniques can combined synergistically to provide superior on-time delivery rates while maximizing energy efficiency. We also present a prototype implementation. Compared to existing protocols our prototype provides higher delivery rates and expends less total energy to do so.

4. REAPER: AN ENERGY-AWARE TRANSPORT PROTOCOL

Given our insights into loss recovery, we designed an energy-aware transport protocol, Reaper, that minimizes energy consumption while maintaining target application reliability requirements using a novel combination of loss recovery mechanisms. By exploiting minimal use of opportunistic dropping, Reaper can more effectively use predictive dropping to minimize the number of reactive drops and increase the application goodput. In this section, we discuss two of Reaper's parameters that control this use of opportunistic and predictive dropping. Along with intelligent dropping, Reaper also supports adaptive applications by exposing relevant information about network cost and availability. Additionally, Reaper optimizes energy consumption using simple application-specific information about data reliability requirements and timeliness. To evaluate the effectiveness of Reaper and its cross-layer design, we implemented and tested a real-world implementation. Our evaluation of this prototype shows the benefits from both Reaper's effective energy conservation techniques and the exchange of cross-layer information. Our results demonstrate the need for such cross-layer information in protocols.

4.1 Loss Recovery Policies

To optimize the fraction of frames that successfully arrive at the receiver while minimizing the unnecessary expenditure of energy, Reaper employs intelligent dropping policies that minimize timing violations and take advantage of opportunistic dropping.

4.1.1 Reactive Dropping

When a fragment of a frame is received past the frame playout time, the receiver signals the sender to drop any unsent fragments remaining in the frame. This signal can be included in an ACK sent by the receiver to minimize overhead. Because reactive drops cannot be triggered until the receiver detects a timing violation, many late fragments will have already been sent, wasting energy. If the detection is too late, the sender will already have sent all fragments, resulting in no energy savings. Therefore, while reactive dropping can be used as a fallback, avoiding reactive drops altogether can save even more energy. Additionally, since the time has already been wasted to send the frame, reactive drops do not help the stream catch up in time, and so may not avoid further late frames.

4.1.2 Predictive Dropping

When a sender predicts that a frame being sent may be late at the receiver, the sender drops any unsent fragments of that frame. Such predictions must be made based on information about the deadline of the frame, as indicated by the application. The specific deadline for a frame depends on the frame rate of the multimedia stream, the inter-frame spacing IFS, and the size of the playout buffer at the receiver, B_p, in seconds.

For each fragment of a frame, it is necessary to estimate its frame's playout time and the estimated arrival time of the last fragment of its frame. To this end, the sender needs the information both from the application and about the current network conditions. From the application, the sender requires B_p, IFS, and the frame number for each fragment. The sender also requires a continuously updated estimate of the network latency L, which can be is estimated from the current roundtrip time ($\overline{RTT}$) as $\frac{\overline{RTT}}{2}$. Because the transmission time is negligible compared to the network latency, an accurate estimation does not require an estimate of the network latency that separates out the full bandwidth component of the roundtrip time.

To estimate the timeliness of a frame it is necessary to predict if its last fragment will arrive on time. If a frame fits in a single packet, then it is only necessary to determine whether the packet containing the frame will arrive on time at the receiver. For a given frame F_i this is equivalent to determining whether the following inequality holds:

$$t + \frac{\overline{RTT}}{2} < t_s + IFS \times i + B_p,$$

where t is the current time and t_s is the time at which the receiver began playout.

However, in practice, video frames are split into multiple fragments, and simple network latency cannot be used to make this lateness prediction. The determination of whether a fragment will be useful depends not on its arrival time, but on the arrival time of the last fragment of its frame. If there are f fragments left to be sent in frame F_i, then for the last fragment to arrive on time, the following inequality must hold:

$$t + (T_f + \frac{\overline{RTT}}{2}) < t_s + IFS \times i + B_p,$$

where T_f is the amount of time to transmit the remaining fragments of the frame.

In the best case, T_f is negligible. In the worst case, each of the $f-1$ fragments after the current one will take as much time to send as the current fragment. These edge cases represent the endpoints of the prediction interval

$$\left[\frac{\overline{RTT}}{2}, f \times \frac{\overline{RTT}}{2}\right].$$

Predictions based on the left-hand side of this interval will be optimistic and may cause fragments to be sent that will arrive late, while those based on the right-hand side will be pessimistic, and may cause frames to be dropped that would have arrived on time. To control the optimism of Reaper's prediction, we provide a parameter ω to shift the prediction within the interval. Setting $\omega = 0$ corresponds to the left endpoint of given interval, and setting $\omega = 1$ corresponds to the right endpoint. Section 4.4.1 presents an evaluation of the effects of the choice of ω on Reaper's performance in terms of the total number of dropped frames.

Thus, estimating whether a frame F_i will arrive on time is equivalent to checking whether the following inequality holds.

$$(i \times IFS) - (t - t_s) + B_p > ((\omega \times \overline{RTT}/2 \times f) + ((1-\omega) \times \overline{RTT}/2).$$

Finally, the sender must estimate the start time of the video playout. To estimate this start time, Reaper must calculate which frame fills the playout buffer and so triggers the stream to start playing. This frame, F_s, can be calculated as follows:

$$F_s \ : \ s \geq \left\lceil \frac{B_p}{IFS} \right\rceil.$$

Given F_s, Reaper can predict t_s based on an estimation of the arrival time of F_s. Any frame sent before the stream starts will arrive on time. Once frame F_s has been sent, Reaper can start predicting whether subsequent frames will be late.

The largest gains in performance, in terms of good frames delivered, are due to predictive dropping. Predictive drops both conserve the energy to send frames that would have been late and allow subsequent frames to be sent sooner, building back some playout buffer at the receiver. Extensive evaluation is given in Section 4.4.

4.1.3 Opportunistic Dropping

Even if the sender estimates that there is enough time to recover a lost fragment, it may not be beneficial, in terms of both energy consumption and average loss rate, to try to recover the frame. This decision can be based on network specific information, such as knowledge about the current level of congestion in the network, or based on application specific information, such as knowledge about the reliability requirements of the application.

To support application reliability requirements, Reaper provides a top threshold (Γ). If the current reliability level is above Γ, lost fragments are never retransmitted and the associated frames are dropped. Disabling such retransmissions has two effects. First, dropping frames that are not necessary to send instead of repairing them through retransmission conserves the energy that would have been spent on the retransmissions. This is the primary function of Γ. The secondary effect of disabling retransmissions is that each frame that is dropped allows the next frame to be sent sooner, potentially preventing that frame from being late as well.

Although the opportunistic drops waste the energy from encoding the frame and transmitting the initial fragments, they frequently prevent energy loss from subsequent frames that would have otherwise been late. Additionally, opportunistic drops allow a stream with very few frames left in its playout buffer to build back up a reserve, preventing large skips in the stream, and reducing the number of predictive and reactive drops needed later.

4.2 Information Sharing Between Layers

Given our energy analysis of the application and network, a small amount of information must be passed between layers to facilitate cross-layer adaptation. This information can be divided into two categories: information that is passed to the transport layer from the application, and information passed to the application from the transport layer. Since Reaper optimizes data transmission based on application information, it is necessary to understand which information needs to be passed between the layers.

From the application, Reaper needs information about frame deadlines and the playout buffer to support predictive dropping. Real-Time Protocol (RTP) [21] can be used to support similar functionality, without interfering with the reliability mechanisms. If the multimedia streams have heterogeneous timing and reliability needs, more sophisticated techniques [9] can be used. Additionally, to support opportunistic dropping, Reaper needs information about the reliability requirements of the application. If the application does not provide a reliability level, Reaper defaults to 100% reliability so that applications that are not Reaper-aware can still function.

Information passed to the application from Reaper represents information about the cost and quality of the data stream. Reaper passes information about the current average number of bytes per good byte sent ($\frac{D''}{D'}$). This, combined with information exposed by the MAC layer about interface costs, allows the application to estimate the cost in terms of energy for sending data. Reaper also informs the application of frame drops so that it can react to the loss

of frames if needed. For example, if the encoder is using a predictive method of encoding, the loss of one frame may imply the loss of following frames that are already encoded based on that frame. Therefore, the sooner the encoder is notified of frame loss, the fewer frames will be lost.

4.3 Prototype

To validate our design, determine the effect of the various parameters, and evaluate Reaper, we implemented Reaper as a user-space library under Linux. The library implementation of Reaper takes data from the applications that links to it and sends it over UDP, with the timing and reliability described as above. A final implementation of Reaper would, of course, be done at the kernel level to take advantage of the decreased packet overhead and system latency.

Reaper uses the TCP New Reno congestion control mechanisms [19], ensuring TCP-friendly behavior. However, the decisions about what data to send and when to retransmit a packet are based on the mechanisms described above. When retransmissions are used, Reaper uses the TCP New Reno's retransmission mechanisms, including timeouts based on estimates of RTT, and the Fast Recovery / Fast Retransmit mechanisms. Since these are well studied mechanisms, we do not go into further detail here. For future work, it would be interesting to develop a rate-based version of Reaper, evaluating the interactions of rate-based mechanisms with the mechanisms presented in Section 3.

4.4 Evaluation

Reaper uses a number of mechanisms to allow it the flexibility to conserve energy while still meeting the needs of the multimedia applications it is built to support. To demonstrate the effectiveness of using these mechanisms together in a single protocol, we use two metrics. First, it is important that the transport layer deliver enough frames to match the application's loss tolerance level. To measure this, we use the concept of application goodput, the number of good frames received on time by the application. Second, the data expansion caused by the transport layer must be kept to a minimum to increase energy efficiency. Therefore, we use data expansion as our energy efficiency metric. Using these metrics, we show that a complete, coordinated combination of the mechanisms presented above results in gains in both energy efficiency and the application goodput over using other subsets of the mechanisms. The greatest gains come from predictively dropping late frames, when combined with opportunistic drops.

For the experiments in this section, our testbed consisted of a Linux laptop running our implementation of Reaper sending data via IEEE 802.11b through a base-station to a wired node one hop away. The application simply sends unencoded data from a file to isolate the effects of using application level information at the transport layer. In Section 5, we present our adaptive application and show that further energy savings can be achieved by using bidirectional cross-layer information to also adapt the application.

4.4.1 *Effects of the Parameters*

Reaper has two parameters that can be used to alter the behavior of the protocol. This subsection analyzes the effects of each one in turn.

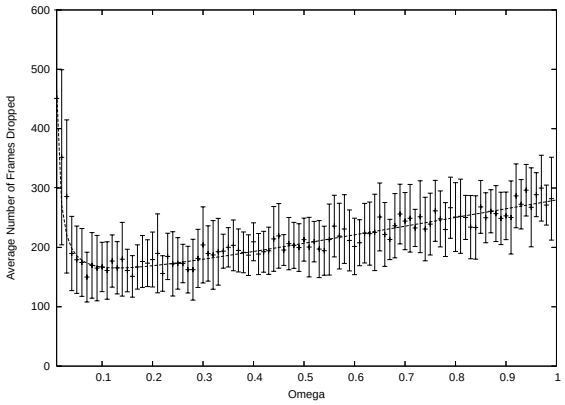

Figure 3: Frame drops as a function of ω, with the mean over 50 trials $\pm$ standard deviation.

To support predictive dropping, ω affects how optimistically Reaper predicts that a frame will arrive on time. If Reaper is over-optimistic, it transmits frames that end up being late at the receiver. This aggressiveness actually results in the successful transmission of more frames because it drops fewer frames that would have actually gotten through. Being over-optimistic, however, results in a *decreased* energy efficiency because the number of frames sent that are late outweighs the extra frames that are pushed through, resulting in a much higher data expansion for the stream. Being over-pessimistic, on the other hand, reduces the number of frames that are actually late to near zero, but causes frames that could have arrived before their deadlines to be dropped. If many of these frames are partially sent before they are dropped, this pessimism can also cause a decrease in energy efficiency. If the frames are all dropped at the beginning of the frame, energy efficiency increases, but the total number of frames received, and therefore the user experience, suffers.

To evaluate the effect of ω on Reaper, we ran a large number of experiments on our real system (see Figure 3). These experiments confirm our intuition and indicate a good operating point of ω. With high values of ω, Reaper is too optimistic about whether there is time to get a frame through. This has the effect of causing many reactive drops, and increases the total number of drops. If ω is too low, on the other hand, Reaper is too pessimistic and predictively drops a large number of frames unnecessarily. We see from Figure 3 that the optimal value for ω should be approximately 0.13. Although Reaper is relatively insensitive to the exact value chosen, values between 0.05 and 0.3 all give a total number of drops within 10% of the optimal value. Choosing a value far from this range, however, can drastically increase the number of frames dropped. For our experiments, we use the optimal value of 0.13. Because these experiments were performed over a wide range of network conditions over the period of a number of days, we have confidence that the omega value chosen will perform well in a wide variety of network conditions.

The use of optimistic dropping is determined by the top threshold (Γ), the maximum reliability desired by the application. This can be seen as the goal reliability level for Reaper. If the reliability level increases above Γ, Reaper

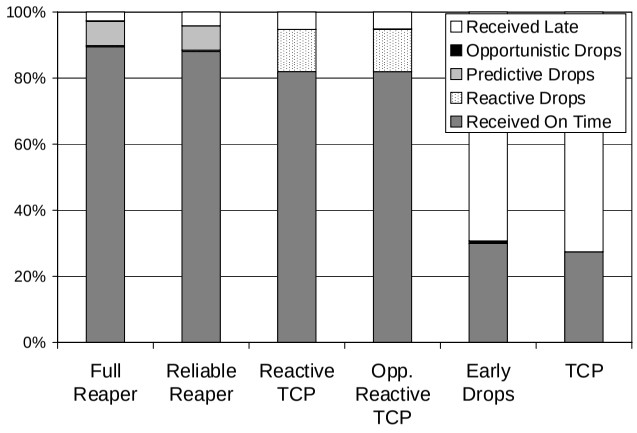

Figure 4: Frames Attempted and Their Disposition

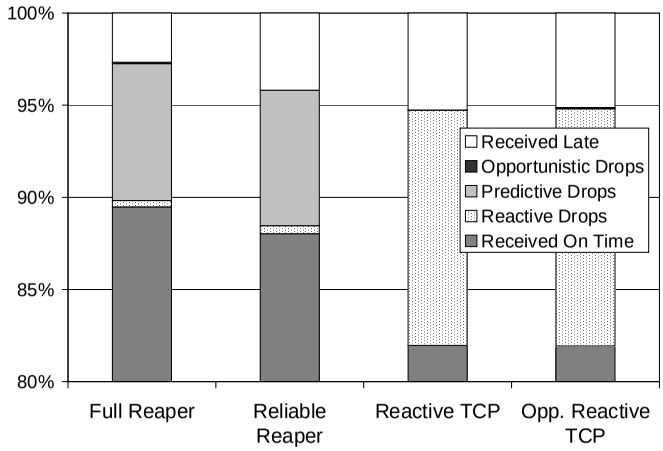

Figure 5: Magnification of Frames Attempted and Their Disposition

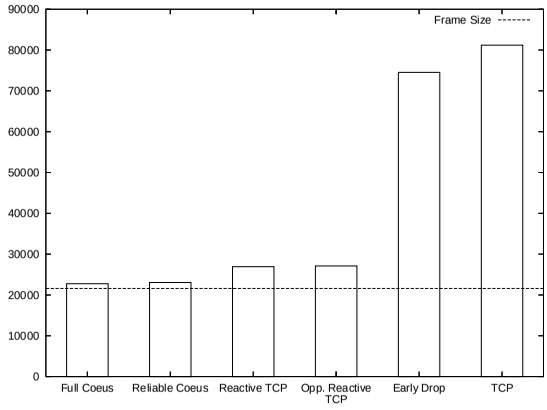

Figure 6: Bytes per Good Frame

disables retransmission of lost packets, and drops the associated frame to conserve energy. Essentially, lower thresholds allow more frames to be dropped during times when the network is in a particularly good state. This has the effect of conserving energy on channels where the effective bandwidth is greater than needed by the multimedia application. The ability of Γ to reduce the number of predictive drops is shown in the next section.

4.4.2 Performance Analysis

To understand the impact of combining the mechanisms used by Reaper, the following analysis presents goodput and energy analysis for six protocols: Reaper, Reaper without opportunistic drops, TCP with reactive drops, TCP with reactive and opportunistic drops, early drop, and TCP. TCP yields full reliability with no concern for timing constraints. Early drop aborts frames if losses are encountered, but has no mechanism to account for late frames. TCP with late frame drop sends all frames reliably but drops frames if notified by the receiver that they will be late. The unreliable variant disables retransmissions when the fraction of frames successfully received is above Γ. Finally, Reaper uses all of the mechanisms described in the previous section, with either full or application-specified reliability. The following results are averages over 50 runs. Each run transfers a 50MB file over an IEEE 802.11b wireless card in a laptop through a base station to a wired receiver one hop away. The wireless conditions and the network load vary over time.

We first evaluate the application goodput for each protocol. As seen in Figure 4, TCP and early drop perform poorly. While TCP gets all of the frames to the receiver, less than 30% of the frames are on time. Early drop performs slightly better, due to its ability to use opportunistic dropping to catch up in the event of a packet loss. However, many frames are still received late. Reliable TCP with reactive dropping performs significantly better, getting roughly 82% of the frames through and on time. This improvement comes from the ability to abort sending frames that are arriving late. TCP with reactive and opportunistic drops performs slightly better (see Figure 5). Reaper's predictive dropping mechanism avoids nearly all reactive drops and increases the goodput nearly 10% over the opportunistic, reactive TCP. Reaper without opportunistic dropping gets approximately 88% of the frames through successfully. However, with opportunistic drops and Γ set to 98%, Reaper gets over 89% of the frames through to the receiver on time and intact (see Figure 5). The majority of the unsent frames are due to Reaper predicting that the frame will be late and skipping ahead, and thus saving the associated transmit energy.

Next, we evaluate the energy efficiency of the protocols. Recall that the energy consumption of a transport protocol is directly affected by the number of bytes used to transmit the data usable by the receiver. Essentially, this is the stream-level data expansion for the protocol. Figure 6 depicts the average bytes sent per good frame for each of the protocols. The line in the graph at about $\sim$22K bytes represents the average number of bytes per frame. Therefore the space above the line represents the average data expansion per frame. As expected, Reaper has a significantly lower data expansion and therefore better energy efficiency compared to the other protocols.

In addition to sending significantly fewer bytes per good byte, Reaper actually sends at least 8% fewer total bytes than any of the other protocols, thereby using less total network energy. It is able to do this through its predictive dropping mechanism: since Reaper does not have to wait

for the receiver to notify it about timing violations, Reaper can avoid sending even the first fragments of a frame which will be late.

However, as discussed in Section 2, the network is not the only, or even necessarily the dominant, component of multimedia system energy consumption. We examine the effect of application CPU energy consumption in the next section, and show how Reaper exposes the information necessary to build applications that can make intelligent decisions about the CPU/network energy tradeoff.

5. APPLICATION ENERGY CONSERVATION

Application energy conservation techniques aim to reduce energy consumption by altering the behavior of the application, which can affect both resource usage and the quality of the data (*e.g.*, reducing the quality of a video stream reduces the amount of data transmitted). Such application adaptations can be based on both energy conservation goals and changes in the availability of resources. For example, Odyssey [10] and Puppeteer [5] use filtering to drop certain parts of a document depending on the amount of bandwidth or the battery lifetime to reduce the network costs (both in terms of bandwidth and energy) for transmission. Both of these solutions adapt the application data to the current network conditions. Essentially, these approaches change the amount of data transmitted based on the bandwidth that is currently available by eliminating parts of the data. They do not need to consider the CPU energy since the adaptation is performed on a proxy. However, for encoding algorithms (*e.g.*, video encoding), the computation necessary to adapt the data stream can result in a significant amount of energy consumption. Therefore, the CPU costs cannot be ignored. Other systems consider adaptations based on dynamic voltage scaling [3] but do not react to network cost fluctuations or are based on expensive network costs [4], but none use bidirectional cross-layer information to minimize system energy.

In the context of lossy encoding algorithms, such as multimedia encoding, there are a number of options for adaptive applications [1, 2, 15]. Fundamentally each type of multimedia encoding allows a tradeoff between the amount of data sent and the quality (*e.g.*, in terms of frames per second) and computation cost for encoding of the data. However, these applications have typically been developed to reduce the amount of bandwidth necessary to transmit data to support performance over low-bandwidth links. The energy increases or decreases due to changes in computation and network costs have not been analyzed. Therefore, such applications must take into account both the number of CPU cycles used and the amount of data created to make adaptation decisions in the face of energy constraints.

5.1 Adaptive Video Encoding Application

To evaluate the use of cross-layer information in the application layer, we developed an adaptive video encoder using the GRACE video encoder library [15], based on H.263 [14]. This video library contains 15 encoding schemes that vary the amount of computation needed to encode each frame and the amount of data produced by the computation for each frame. Therefore, to determine the most efficient encoding scheme, an adaptive application needs information about the per-byte network energy costs and the per-cycle CPU energy costs.

The challenge to choosing the correct encoding scheme is that the choice of encoding scheme is driven E_{trans}, which fluctuates with changes in $\frac{D''}{D'}$ and changes in R. To capture these changes, we define Ω_{net} as follows:

$$\Omega_{net} = \overline{\frac{D''}{D'}} \times \frac{P_{card}}{R},$$

where $\overline{\frac{D''}{D'}}$ is the average data expansion for the transport layer and R is driven by network conditions, which fluctuate with changes in noise in the channel. In addition to Ω_{net}, the application also needs the estimated CPU cost per-cycle (Ω_{CPU}) defined as follows:

$$\Omega_{CPU} = \frac{P_{CPU}}{F}.$$

To determine the expected energy consumption for a given encoding scheme ($E^{(m)}$), where m is the encoding scheme, let D be the size of the unencoded data, $D'^{(m)}$ be the size of the data after encoding with scheme m, and $C^{(m)}$ be the number of cycles needed to encode the data using encoding scheme m. If no encoding is used ($m = 0$), the energy consumed is driven purely by the network costs:

$$E^{(0)} = D \times \Omega_{net}.$$

When encoded data is used, the energy consumption must include the computation cost and the effect on the reduction in the data transmitted:

$$E^{(m)} = (D'^{(m)} \times \Omega_{net}) + (C^{(m)} \times \Omega_{CPU}). \quad (1)$$

Therefore, the application should send unencoded data when $E^{(0)} < E^{(m)}$, for all m; otherwise, the application should choose the encoding level that minimizes $E^{(m)}$.

Across the frames in a stream, the performance of any encoding scheme varies, in terms of the encoding efficiency as well as the number of cycles needed to encode the data. Therefore, a prediction of the encoding efficiency and the number of cycles needed for each encoding scheme can be used to determine the most energy-efficient scheme. Since, for many videos, the encoding efficiency of the stream is fairly smooth. A standard weighted moving average can be used to update the predictions for both encoding efficiency and number of cycles. The update equation for cycle count is:

$$C^{(m)} = (\alpha \times C^{(m)}) + ((1 - \alpha) \times C),$$

where $C^{(m)}$ is the prediction for cycle count for scheme m. A similar approach can be used for $D'^{(m)}$.

Since no information is known about C or D' for any of the schemes before a stream starts, each encoding scheme is probed during the transmission of the first M frames, where there are M encoding schemes available, initializing $C^{(m)}$ and $D'^{(m)}$. The cost of this probing comes from using the less efficient schemes during the the initialization period.

```
ApplicationAdapt()
 1  MAXLEVEL : number of encoding schemes (M)
 2  data : data frame
 3  start, stop : cycle counts
 4  next : encoding efficiency
 5  data_size ← sizeOf(data)
 6  if initialize
 7     then count(start)
 8          encode(data, next)
 9          count(stop)
10          if next++ > MAXLEVEL
11             then initialize ← false
12     else next ← Cost(noEncode)
13          for i ← 1 to MAXLEVEL
14          do if next > Cost(i)
15                then next ← i
16                     count(start)
17          encode(data, next)
18          count(stop)
19  updateCycleHistory((stop − start), next)
20  updateEffHistory(data_size, sizeOf(data), next)
```

Figure 7: Application Adaptation Algorithm

During run-time, Equation 1 is used to determine $E^{(m)}$ for $1 \leq m \leq M$ to determine the optimal encoding scheme to be used. Since the encoding efficiency of a given encoding scheme can vary, predictions developed from one video are not useful for other videos, which is the reason for initializing all values at the beginning of each new stream.

To prevent a particularly bad efficiency or cycle count from eliminating a particular scheme completely, the history of each unused scheme is decayed by a small percentage, essentially making them cheaper and causing them to be probed after a long time of inactivity.

5.2 aVE Experimental Setup

aVE is implemented as an application on top of Reaper using the adaptation algorithm presented in Figure 7. Our testbed consists of a laptop, an IEEE 802.11b base station, and a wired host. The wired host is the receiver and runs Fedora Core 4 with the user-space implementation of Reaper. The laptop acts as the sender and also runs Fedora Core 4 and Reaper. The laptop has an IEEE 802.11b interface and a Pentium M processor. The processor runs at 600MHz and $P_{CPU} = 3W$. $P_{card} = 980mW$ and the MAC layer dynamically adapts its bit rate, R, according to the channel conditions. For experimental purposes, we set the card to a fixed R to evaluate the long term effects of different CPU and network costs. To determine the energy consumption, the CPU rate and cost are available via the Linux `/sys` file system. For these experiments, the receiver is one wired hop away from the base station. All experiments are run on an active wireless network with uncontrolled cross traffic. Therefore, our experiments represent real-world scenarios where a wireless host must compete for channel time and network bandwidth.

5.3 aVE Evaluation

Without cross-layer information about network and CPU costs and bandwidth availability, an adaptive application cannot be optimized to save energy. However, our evaluation show that aVE coupled with Reaper can effectively adapt to the dynamics of network cost and availability. To demonstrate this, we evaluate aVE using a standard suite of video data. The compressibility of the data varies across videos and across frames within a single video. The video files are CIF format and frames are fragmented into one to thirty-five 1440B fragments depending on the encoding scheme chosen by the application.

For these experiments, the relative cost of the CPU and the network depends on the current rate at which the network is operating. For low rates, the network is quite expensive compared to the CPU, while for the highest rate the network is cheaper than the CPU. Therefore, no one fixed encoding scheme is likely to be optimal for all network conditions. To show this, we compare each fixed encoding scheme with aVE for each of the network bit rates (1Mbps, 2Mbps, 5.5Mbps, and 11Mbps). We will show that there is no best fixed encoding scheme as the network conditions change, supporting our claim that the application needs cross-layer information to optimize for energy efficiency.

The graphs in Figure 5.3 show the total system energy consumption for each encoding scheme at each bit rate. When the network is running at 1Mbps, encoding schemes 0 through 7 use a considerable amount of CPU energy and achieve a large amount of data compression. On the other hand, schemes 9 through 14 use very little CPU and achieve little data compression. Scheme 8 employs a technique that uses moderate CPU energy and achieves moderate compression. However, for 1Mbps, scheme 6 uses the minimum system energy. Our adaptive algorithm, aVE, uses less than 5% more energy than this best fixed scheme. This overhead is due to the need to probe the highly inefficient compression schemes. When the network rate is increased to 2Mbps, the best fixed compression scheme is no longer 6 but shifts to 7 because the network is becoming cheaper. Scheme 6, however, still uses slightly less energy than aVE (see Figure 5.3b). However, when the network operates at 5.5Mbps, scheme 6 uses slightly more energy than aVE (see Figure 5.3c). Scheme 7 is still the best fixed encoding scheme. Finally, when the network is operating at 11Mbps, network energy is dominated by CPU energy. Therefore, the encoding schemes that use the smallest amount of CPU use the least amount of energy, as shown in Figure 5.3d. In this case, schemes 6 and 7, which used the least energy at lower network speeds, use 40% and 20% more energy, respectively. Scheme 14, which is the best fixed in this instance, uses as much as 350% more energy than aVE at lower network speeds. In all cases, aVE uses less than 12% more energy than the best fixed scheme. Given encoding schemes or network technologies with greater cost variance, our adaptive system would perform even better compared to the best fixed schemes.

Since there is no best fixed compression scheme, choosing any one fixed compression scheme will result in poor energy efficiency depending on the state of the network. Without cross-layer information to support application scheme adaptation, energy efficiency is impossible to attain, especially in the face of dynamic network characteristics. By using information both about the prevailing network conditions and about current CPU costs, aVE intelligently adapts its encoding schemes and successfully conserves energy in all environments. While some cross-layer information is neces-

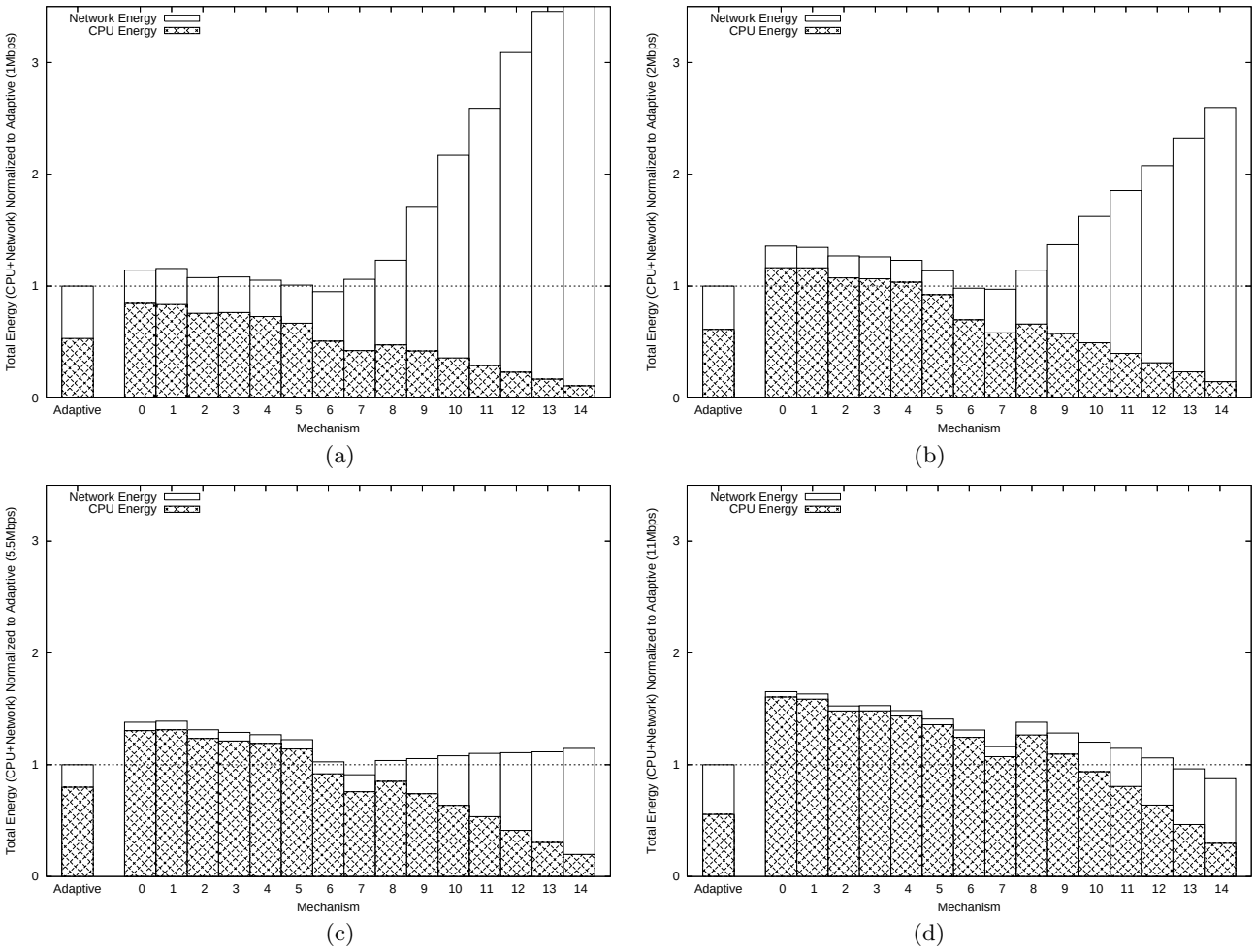

Figure 8: Energy Performance at (a) 1 Mbps, (b) 2Mbps, (c) 5.5Mbps, (d) 11Mbps

sary, the information required is simple, allowing easy development of adaptation algorithms.

6. CONCLUSIONS AND FUTURE DIRECTIONS

Although it is well-known that applications need to adapt to changes in bandwidth to optimize performance, optimizing for energy efficiency requires knowledge of the relative cost of the application's computation and the transmission of the resultant data. While some research has investigated pure application-level adaptations or pure network-level adaptations, our research shows that bi-directional information sharing between the application and the transport layer is necessary to minimize energy consumption. Based on our energy model that exposes interdependencies between the application and the transport layer, we examine the energy efficiency of transport level reliability mechanisms and their impact on application performance and total system energy consumption.

Given the benefits of bi-directional information sharing, we design, implement, and evaluate an energy-aware transport protocol, Reaper, that leverages information about the application's timing and reliability requirements to drive the effective use of reactive, predictive, and opportunistic dropping mechanisms. Through our evaluation of Reaper, we demonstrate that the benefits of the combination of the three dropping mechanisms outweighs the benefits from using any one subset of the three mechanisms.

To demonstrate the full benefit of bi-directional information sharing, we design and evaluated an adaptive video encoding application, aVE, that is built on top of Reaper. Because of its simple design and on-line probing, aVE approaches the efficiency of the best encoding level and in the worst case uses only slightly more energy than the oracular best fixed encoding mechanism regardless of network rate. In the common case, where the network fluctuates between data rates, aVE performs significantly better than any single fixed level.

This paper focused on the effects of dynamic network costs and conditions. Future work includes the integration of variable CPU energy consumption via dynamic voltage scaling. This requires sharing additional information between the layers via the mechanisms already developed. Along the same lines, other wireless technologies such as ultra-wide

band and other low-power CPU platforms expand the available adaption space.

With respect to the specifics of Reaper, it would be interesting to explore the impact of alternative loss-notification algorithms, such as ELN [20] and SACK [18]. Such algorithms provide more feedback to the transport layer allowing more intelligent loss-recovery decisions as well as more accurate feedback to the application.

Finally, a kernel-level implementation of Reaper and additional applications would allow more comprehensive testing and evaluations of the our adaptive algorithms and protocols. For example, an application for reliably transferring files with various degrees of compression could minimize the energy used and extend mobile system lifetimes.

7. REFERENCES

[1] Mpeg video compression. "http"//www.mpeg.org/".

[2] Speex audio codec. "http://www.speex.org/".

[3] S. V. Adve, A.F. Harris, C.J. Hughes, D.L. Jones, R.H. Kravets, K. Nahrstedt, D. G. Sachs, R. Sasanka, J. Srinivasan, and W. Yuan. The illinois grace project: Global resource adaptation through cooperation. In *SHAMAN*, 2002.

[4] M. Corner, B. Noble, and K. M. Wasserman. Fugue: Time scales of adaptation in mobile video. In *Proc. of the SPIE Multimedia Computing and Networking Conference*, 2001.

[5] Eyal de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *3rd USENIX Symposium on Internet Technologies and Systems*, 2001.

[6] L. Donckers, P. Havinga, G. Smit, and L. Smit. Enhancing energy efficient tcp by partial reliability. In *13th IEEE PIMRC*, 2002.

[7] Gavin Holland, Nitin Vaidya, and Paramvir Bahl. A rate-adaptive mac protocol for multi-hop wireless networks. In *Proc. 7th ACM International Conference on Mobile Computing and Networking (MobiCom '01)*, 2001.

[8] R. Kravets, K. Calvert, P. Krishnan, and K. Schwan. Adaptive variation of reliability. In *HPN 1997*, 1997.

[9] Jia-Ru Li, Sungwon Ha, and Vaduvur Bharghavan. Hpf: A transport protocol for supporting heterogeneous packet flows in the internet. In *IEEE INFOCOM*, 1999.

[10] Brain Noble. System support for mobile, adaptive applications. *IEEE Personal Communications*, 2000.

[11] C. Papadopoulos and G. Parulkar. Retransmmission-based error control for continuous media applications. In *NOSSDAV*, 1996.

[12] Sassan Pejhan, Mischa Schwartz, and Dimitris Anastassiou. Error control using retransmission schemes in multicast transport protocols for real-time media. *IEEE/ACM Transactions on Networking*, 1996.

[13] Daji Qiao, Sunghyun Choi, Amit Jain, and Kang G. Shin. Miser: An optimal low-energy transmission strategy for ieee 802.11a/h. In *Mobicom*, 2003.

[14] Telenor Research. Tmn (h.263) encoder/decoder, version 2.0. http://www.xs4all.nl/ roalt/h263.html.

[15] D.G. Sachs, S. Adve, and D.L. Jones. Cross-layer adaptive video coding to reduce energy on general-purpose processors. In *Proc. on the International Conference on Image Processing (ICIP'03)*, 2003.

[16] J. Seo, N. Sung, S. Lee, N. Park, H. Lee, and C. Cho. An adaptive type-i hybrid-arq scheme in a tdma system over a non-stationary channel. In *First Workshop on Resource Allocation in Wireless Networks*, 2005.

[17] R. Sinha and C. Papadopoulos. An adaptive multiple retransmission technique. In *NOSSDAV*, 2004.

[18] IEEE Computer Society. Tcp selective acknowledgement options, rfc 2018, October 1996.

[19] IEEE Computer Society. Tcp rfc 2581, April 1999.

[20] IEEE Computer Society. The addition of explicit congestion notification (ecn) to ip rfc 3168, September 2001.

[21] IEEE Computer Society. Rtp rfc 3550, July 2003.

[22] B. Wah, D. Lin, and X. Su. A survey of error-concealment schemes for real-time audio and video transmission over the internet. In *Proc Int Symposium on Multimedia Software Engineering*, 2000.

[23] Michele Zorzi and Ramesh R. Rao. Error control and energy consumption in communications for nomadic computing. *IEEE Transactions on Computers*, 46(3):279–289, 1997.